**Project Summary and Reflections Report**

**Summary**

While adhering to requirements is critical, it is also necessary to be creative and explore boundary conditions, edge situations, and unforeseen events. This aids in the discovery of hidden concerns that may not be directly stated in the requirements. The testing strategy should strike a balance between ensuring compliance with requirements and investigating the system's behavior in a variety of circumstances. I think my testing approach tested all possible test scenarios. For instance, the requirements stated that certain fields like the ids, first name, last name, phone number, task name, and task description should not be longer than a certain length and should not be null “ContactService” and “TaskService” projects. So, I tested each of the fields with longer than specified lengths and null values to see if the JUnit test succeeds. Low and behold, it succeeded. My JUnit test coverage on the classes being tested for both the “ContactService”, “TaskService”, and “AppointmentService” projects came out to be 100% attesting that my test coverage on the objects being tested was sufficient, complete, and comprehensive.

My code was technically sound based on the data validation tested on the entity classes in the JUnit test. For example, in the JUnit testing classes for all the projects, “ContactService”, “TaskService”, and “AppointmentService”, the testing unit called “dataValidations()” passed all the test cases tested for like checking whether a field was longer than the requirements or null. Moreover, my code was absent of syntax errors and clearly organized to ensure it had the best clarity, code quality, and reliability. Not only that, but my code for the projects also tested positive for efficiency as my JUnit tests successfully ran in 0.133 to 0.157 seconds and completed 5/5 runs with multiple units being tested. You can see the screenshots of the tests ran on all the projects below including the run data and the coverage.

A screenshot of a computer program

Description automatically generated

Figure 1. Shows the test coverage on the Task and TaskService classes.

A screenshot of a error message

Description automatically generated

Figure 2. Shows the run data for the TaskTest and TaskServiceTest JUnit testing.

**A screenshot of a computer

Description automatically generated**

Figure 3. Shows the test coverage on the Contact and ContactService classes.

A screenshot of a error message

Description automatically generated

Figure 4. Shows the run data for the ContactTest and ContactServiceTest JUnit testing.

A list of words on a white background

Description automatically generated

Figure . Shows the test coverage on the Appointment and AppointmenService classes.

A screenshot of a computer error message

Description automatically generated

Figure . Shows the run data for the AppointmentTest and AppointmentServiceTest JUnit testing.

**Reflection**

Going through the different phases and milestones of the project, I found unit testing to be the most suitable and effective testing technique or strategy in my projects.Unit testing is a software testing technique that isolates specific units or components of a software program from the rest of the code. A unit is the smallest tested component of a program, which is typically called a function or method. Unit testing ensures that each unit of code performs as intended and meets its design standards and needs. Unit tests are often automated, which implies that testing frameworks can perform them automatically. They are written by developers (me) to ensure that their code works properly. If a unit test fails, it means there is an issue in the code that needs to be fixed.

I chose the most popular, standardized, and easy-to-use unit testing framework in Java called JUnit. JUnit is widely recognized as the best framework for unit testing in Java, and it has acquired this distinction for a variety of compelling reasons. For starters, it stands out for its simplicity and ease of usage. JUnit's easy, annotation-based approach makes it simple for developers to build and execute tests with minimal setup. Its status as the Java ecosystem's unit testing standard also adds to its importance. This extensive usage guarantees that developers are well-versed in it, and it is well-supported by a wide range of IDEs and build tools. Furthermore, JUnit's smooth interface with popular Java IDEs such as Eclipse, IntelliJ IDEA, and NetBeans enables convenient test execution right within the development environment. Moreover, the fact that it is mature and well-established, as well as having a vast community of users and contributors, attests to its dependability and stability. JUnit includes a variety of features that improve the efficiency and efficacy of unit testing, including support for parameterized tests, built-in assertion methods for result validation, and the ability to organize test cases into suites. It also smoothly interfaces with Continuous Integration platforms like Jenkins, allowing for automated testing as part of the build process. These characteristics, when combined with a robust ecosystem of tools and community support, make JUnit the premier choice for unit testing in Java, ensuring high-quality, error-free code.

Unit testing with JUnit is essential in software development across a wide range of projects and circumstances. It enables developers in agile development settings to continuously validate individual units or components of code, ensuring they perform as intended. This is especially important in large-scale projects with complicated codebases, because detecting and correcting issues early in the development process can save money on delays and rework. Furthermore, JUnit's support for parameterized tests enables complete testing of many input circumstances, increasing test coverage and revealing potential edge cases that might otherwise go unnoticed. JUnit's robustness and mature framework provide the required assurance that critical functionality performs flawlessly in safety-sensitive industries like healthcare or aviation, where software reliability is paramount. The automated nature of JUnit tests in Continuous Integration pipelines ensures that new code does not generate regressions in projects with frequent code changes or contributions from different developers. This protects against unwanted consequences and ensures code integrity. Furthermore, JUnit's compatibility with a variety of Integrated Development scenarios (IDEs) and Continuous Integration technologies makes it versatile and adaptable to a variety of project scenarios. In conclusion, JUnit's ability to validate code at the unit level, support for various testing scenarios, and seamless integration into various project workflows make it an invaluable tool for ensuring software quality and reliability across a wide range of development projects and situations.

We left out Integration testing based on time limitations and requirements; however, it is another useful and important software testing technique for any project, besides unit testing. Integration testing is another useful and important software testing technique, besides unit testing, in software development in which individual system modules or components are joined and tested as a group. The primary purpose is to validate the interactions and interfaces between these elements to ensure they work as planned. This strategy is critical since software is often built from numerous smaller elements, and it is critical to ensure they can communicate and function together flawlessly. Integration testing aids in the discovery of problems with data transmission, communication protocols, and interface compatibility. Developers can uncover and correct potential conflicts or discrepancies that may not be apparent when modules are tested in isolation by running these tests. This procedure ultimately improves the software's overall dependability, stability, and performance, lowering the risk of post-deployment issues and assuring a more pleasant end-user experience. Moreover, another software testing strategy that we would have employed would be Compatibility testing. Compatibility testing is an important software technique in software testing because it assures that a piece of software works as intended across several platforms, browsers, operating systems, and devices. Its primary goal is to ensure that the software is compatible with a wide range of setups and that it provides a consistent and dependable user experience independent of the user's environment. This type of testing aids in the discovery of any compatibility concerns, such as layout differences, feature failures, or performance changes. Developers can address any cross-platform concerns proactively by undertaking compatibility testing, ensuring that the product satisfies the different needs of its user base. As a result, the product becomes more inclusive and user-friendly, allowing it to appeal to a wider audience.

As a software engineering professional, being rigorous in your dedication to quality is critical for several reasons. To begin, high-quality code is the foundation of dependable software. It guarantees that programs work as they should, fulfilling user expectations and preventing costly errors or malfunctions. Furthermore, it leads to a great user experience, which is an important aspect of user happiness and retention. Furthermore, well-crafted code is easier to maintain, debug, and extend, eventually saving time and money in the long term. Cutting corners in code authoring or testing can result in the accumulation of technical debt, which can be damaging to the long-term viability of a project. Technical debt results from quick fixes or inefficient practices that, although offering short-term benefits, have long-term consequences. For example, ignoring adequate error handling may result in unpredictable behavior down the road, making it more difficult to diagnose and fix problems. Similarly, failing to do rigorous testing can allow flaws to get through, potentially causing major issues in production. It is critical to take a proactive strategy to avoid technical debt. Conducting code reviews with peers, for example, might uncover possible issues early on, guaranteeing adherence to best practices. Code reviews would eliminate a lot of the bias present when testing one’s own code (my code) as something not so big to me may raise flag for another person or developer reviewing/testing the code that may have been missed if I was the sole tester. Automated testing, such as unit, integration, and regression tests, helps to protect against unintended consequences of code modifications. Moreover, keeping clear and comprehensive documentation helps in understanding and maintaining the codebase. Regularly examining and rewriting code to remove redundancies, improve readability, and improve speed minimizes debt growth. Finally, remaining current on industry trends and best practices helps you prevent using old or inefficient coding techniques. By implementing these tactics, a software engineering professional can maintain a strong commitment to quality while avoiding the risks of technical debt. To conclude, it is vital to be thorough and comprehensive when it comes to testing software as you are the last line before it makes it to the public, production, etc. There have been many cases where software failure caused many fatal outcomes in the real world once deployed. Thus, software testing should not be undermined and should be done with enough time at hand.

**References**

Jakubiak, N. (2022, December 6). *JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests*. Parasoft. Retrieved from <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing : An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.